Managing OpenBiblio Plugins

Contents

OpenBiblio has a very flexible plugin system. This document will describe how you can enable and disable plugins, understand the plugins that are included with OpenBiblio, and design a new plugin.

Enabling plugins

Tip
You will need access to the Tools menu.

1. Go to Tools → Plugin Manager.
2. Make sure that Plugins are Allowed.
3. Check the box next to each plugin you are interested in using.

Plugins bundled with OpenBiblio

These plugins all come bundled with a default OpenBiblio installation.

CSS Utilities (cssUtils)

This plug-in will looks for two categories of CSS issues:

1. CSS entries in . . . /styles.css that appear to not be in use
2. CSS classes referenced in OB that do not appear in . . . /styles.css

WARNINGS

No fixes are suggested or made. The user should be VERY careful in removing any that appear to not be in use as some css usage is dynamic via jQuery and is dificult to detect programatically.

Query selectors having multiple entries (e.g. $(*.reqd sup*)) are not currently parsed as multiple entries to be searched.

Pull-Down List Manager (List Manager)

This plug-in is intended to provide a developer with a simple means to view the contents of pull down lists that are used in various locations throughout OB.

No means are provide at this time to modify these lists; use phpMyAdmin or equiv for that purpose.

Media Fields

This plug-in is intended to provide a means for OB users to exchange input form layouts.

Some media forms are infrequently used and a new user may not know just what material should be collected.

Using this plugin, it is possible to create a text file of a layout that can be emailed to another for them to import.

Orphan File finder (orpahnFiles)

This plug-in is intended to find old abandoned files. The concept here is to find files which:

1. are not referred to via any menu
2. are not referred to by any HTML mechanism such as Javascript, CSS, Form Actions, Images, etc.
3. are not referred to by any PHP mechanism such as 'includ'e s or 'require's
4. are not referenced in Object class inheritances.

Note
there are many legacy files and folders the have an initial letter'x'. These are to be considered *Legacy* works and may be removed from the OB project at the Lead Developer's descretion.

Note
There is an entire .../docs folder that has never been integrated into OB ver 1.0. this folder may be removed from the OB project at the Lead Developer's descretion.

Translation Utilities (transUtils)

This plugin is intended as an aid to the creation or maintenance of a locale translation. All T(...) entries in OB .php files are checked. Each available translation is tested seperately.

The following functions are available:

1. Check for Duplicate Entries. Intended to find duplicate entries which may be out of place alphebetically and so un-noticed.
2. Check for Unused Entries. Look for table entries which have been abandoned, or through miss-spelling somewhere, are not currently being used.
3. Check for Needed Entries. Look for T(...) entries which do not have a corresponding entry in the trans.php file.
4. Check for Potential Entries. Look for quoted strings (both normal display, and error) in all files which do not have a T(...) surround. Error messages embedded in the various language processors (PHP, JS, CSS, mySQL, etc) will not be caught.

Call Number Utilities (CallNoUtils)

This section describes the Call Number Utilities plugin.

A pull-down allows a search for:

1. items without an assigned call number
2. proposed call numbers based on a desired schema
3. posting proposed call numbers as needed.

Find Records without call numbers

This function will scan the entire database and return a list of all biblios which do not have a call number assigned.

Dry Run of call number add

This function will propose call numbers based on the preferred source.

Add call numbers to records

This function will post the proposed call numbers.

Creating a new plugin

This requires basic PHP skills and access to the files.

Design the plugin

1. Create a new directory that begins with the word plugin, followed by an underscore, followed by the name of your plugin. In UNIX systems, you might type something like this:

   ```
   mkdir plugin_excitingPlugin
   ```

2. If you need to include custom CSS or JS, add it to a file called custom_head.php in this new directory.
3. Create three PHP files in the directory: a JS file, a Srvr file, and a Forms file.

Create a plugin form

Your users will interact with your plugin using a form, located at `plugin_excitingPlugin/excitingForms.php`.

A basic plugin consists of a require statement to get the necessary methods, some code establishing basic UI, some HTML form markup, and a section to display results and messages. Here's a simple example:

```
<?php
  require_once("../shared/common.php");

  $tab = "tools";
  $nav = "ExcitingMgr";

  require_once(REL(__FILE__, "../shared/logincheck.php"));
  Page::header(array('nav'=>$tab.''.$nav, 'title'=>''));

?>
<h1 id="pageHdr" class="title"><?php echo T("excitingMgr"); ?></h1>
    <section>
        <fieldset id="exciting">
            <label for="site_cd" >Choose your favorite library:<select id="site_cd" ></ ↩
                select></label>
            <input type="button" id="showResultsBtn" value="Show results" />
        </fieldset>
    </section>
    <div id="rslts"></div>

<?php
require_once(REL(__FILE__, "excitingJs.php"));
?>
```

Create a plugin JS file

Your plugin's JS file — located at `plugin_excitingPlugin/excitingJs.php` — is responsible for filling your Form with dynamic data, whether it is filling drop-down menus or presenting the results of your query from the server.

A basic JS file consists of functions that capture data from the user and send it to the server file(s). Here's a simple example:

```
<script language="JavaScript" defer>
"use strict";

var excite = {
        init: function () {
                excite.url = 'excitingSrvr.php';
                excite.listSrvr = "../shared/listSrvr.php";
                excite.fetchSiteList();

                $('#showResultsBtn').on('click',null,excite.doSendUserData);
        },
        fetchSiteList: function () {
                $.getJSON(excite.listSrvr,{mode:'getSiteList'}, function(data){
                var html = '';
                for (var n in data) {
                    html+= '<option value="'+n+'">'+data[n]+'</option>';
                }
                $('#site_cd').html(html);
                });
        },
        doSendUserData: function () {
                $.post(excite.url, {'mode':$("#site_cd").val()},
                        function (response) {
                                $('#rslts').html(response);
                                $('#rslts').show();
                });

        },
};
$(document).ready(excite.init);

</script>
```

Create a plugin server

Your plugin's server — located at `plugin_excitingPlugin/excitingSrvr.php` — is responsible for three tasks:

1. Performing any actions the user requests.
2. Obtaining any relevant data from the database, file structure, or external API.
3. Preparing a nice HTML display of the relevant data, which your JS file will add to the Form to display to your user.

A basic plugin server consists of a require statement to get the necessary data and methods and a switch statement to take care of the `$_POST['mode']` variable. Here's a simple example:

```
<?php
        require_once('../shared/common.php');
        switch ($_POST['mode']){
                case 1:
                        echo 'That\'s my favorite library too!';
                        break;
                case 2:
                        echo 'That library is pretty good.';
                        break;
                default:
                        echo T('invalid mode');
                        break;
        }
```

Note
If you would like to follow OpenBiblio style and serve up JSON from your Server to be processed by your JS file, be sure to use POST requests, rather than GET requests, to avoid the potential of Javascript Hijacking.

If you would like to access data from the database, you should require the appropriate model from the model directory. For example, if you'd like to use the Biblios model, you would include this in your plugin_excitingPlugin/excitingSrvr.php:

```
require_once('../model/Biblios.php');
$bib = new Biblios;
```

You can then access all those great methods and data through the `$bib` object.

Add your plugin to the OpenBiblio menu

1. Create a new directory inside the plugins directory with the name of your plugin. In UNIX systems, you might type something like this:

   ```
   mkdir plugins/excitingPlugin
   ```

2. Create a file within this new directory called nav.nav

   ```
   <?php
   Nav::node('tools/excitingPlugin', 'Exciting Plugin', '../plugin_excitingPlugin/ ↩
       excitingForms.php');
   ```

3. If you'd like to use Obib's localization framework but need to localize some more terms, create a file called tran.tran. You can use this to add new terms to your locale.

   Tip
   Plugins already have all of the exciting localization ability built in. However, note that you cannot override localized term definitions within tran.tran.

   

   Warning
   There is not yet a place to specify localized terms for plugins.

4. Turn on your plugin using Tools → Plugin Manager.